CMPS 312

Android Fundamentals

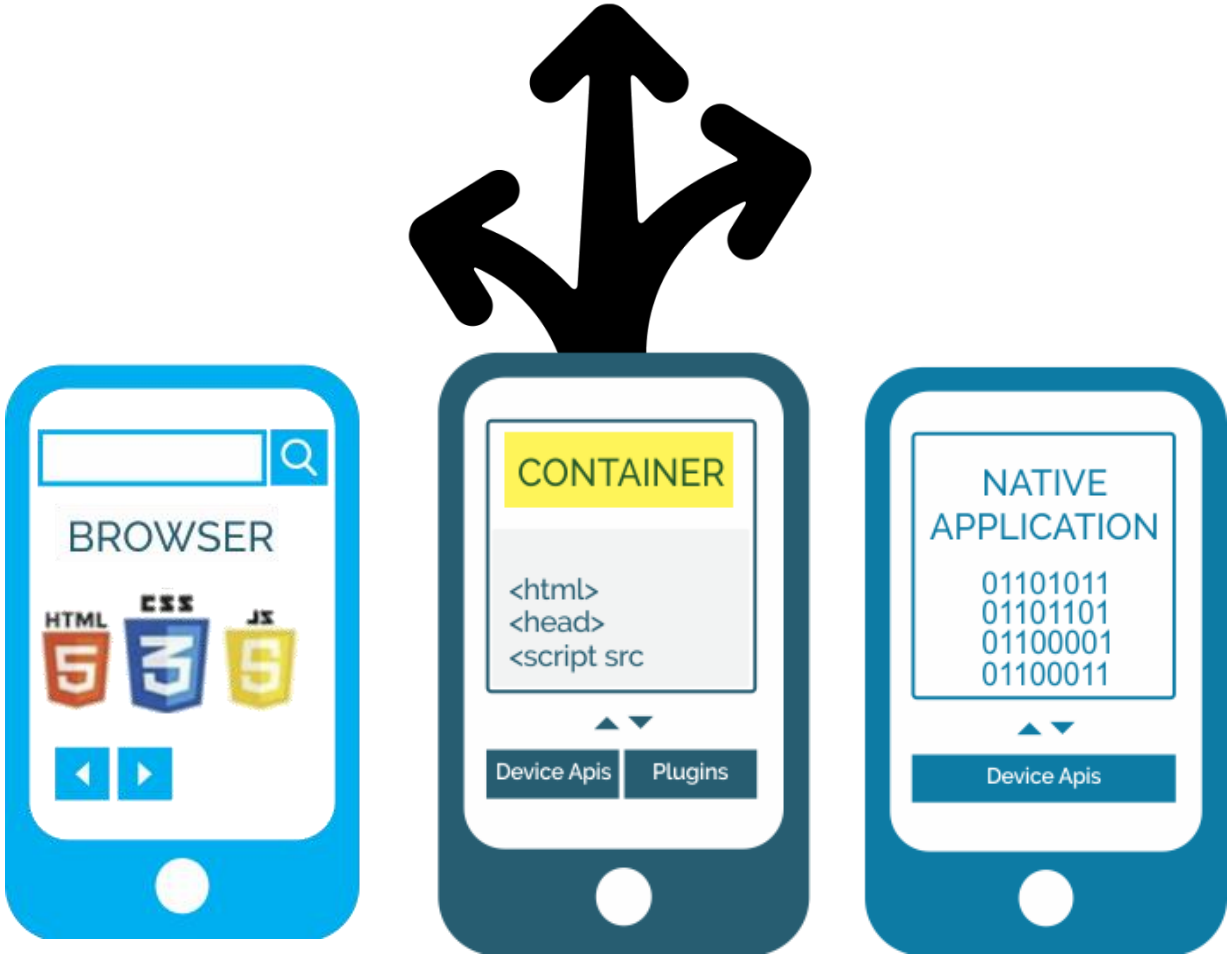
Dr. Abdelkarim Erradi

CSE@QU

Outline

1. Mobile Development Approaches
2. Introduction to Android
3. Imperative UI vs. Declarative UI

Mobile Development Approaches

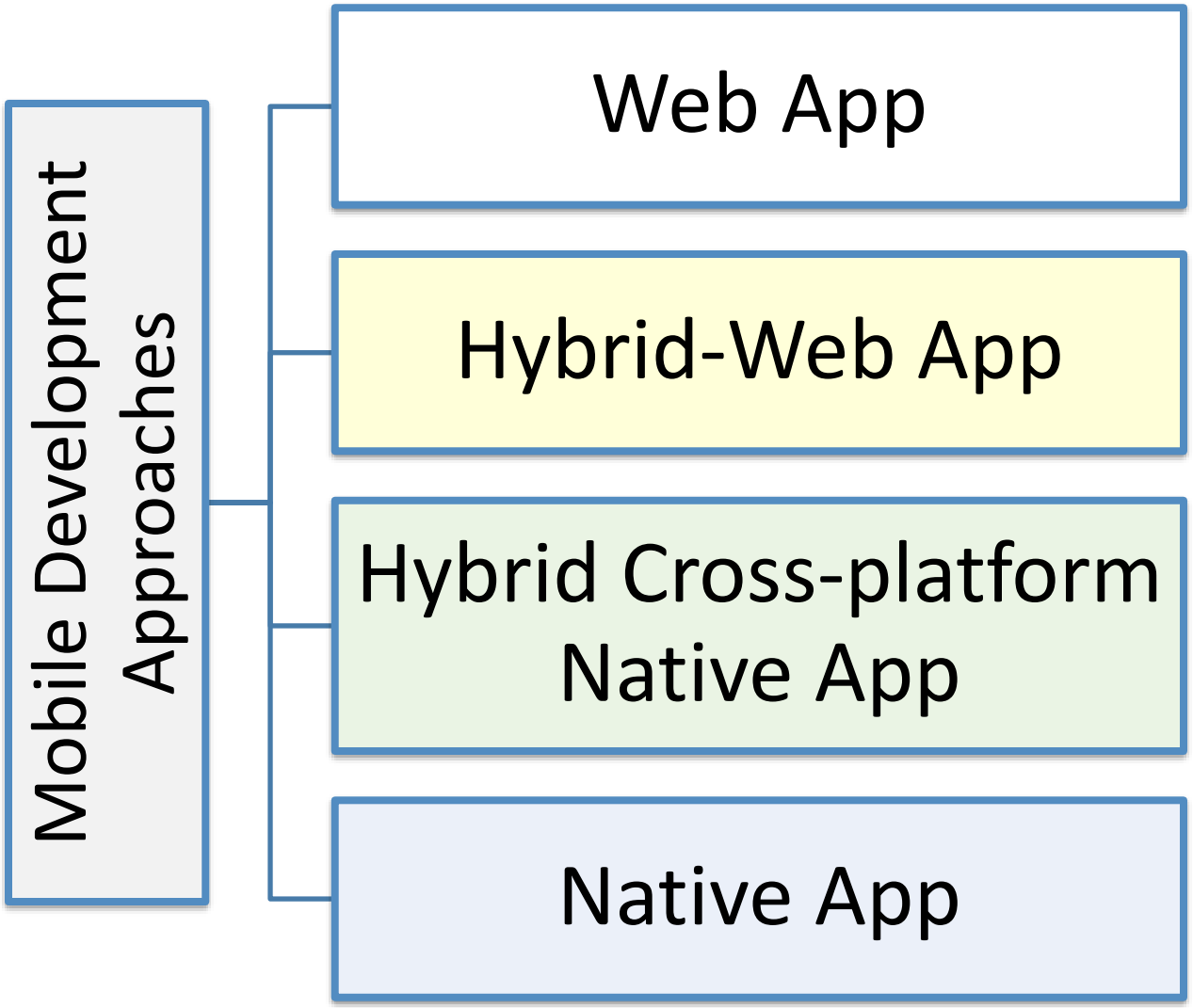


Back

Why learn app development?

- Smart devices are ubiquitous
  - Estimated 3.5 billion smartphones + tablets, smart watches, IoT devices...
  - Apps interwoven into daily life – work, play, study
  - Mobile = **dominant** end-user device. It represents and intimately "knows" the user: much more than just a PC, it represents the user
  - Connected to the outside world: **sensing**, **location**, **communication**
- Apps less expensive and more portable
- Large market opportunity for businesses and developers

Mobile Development Approaches



Web App Development

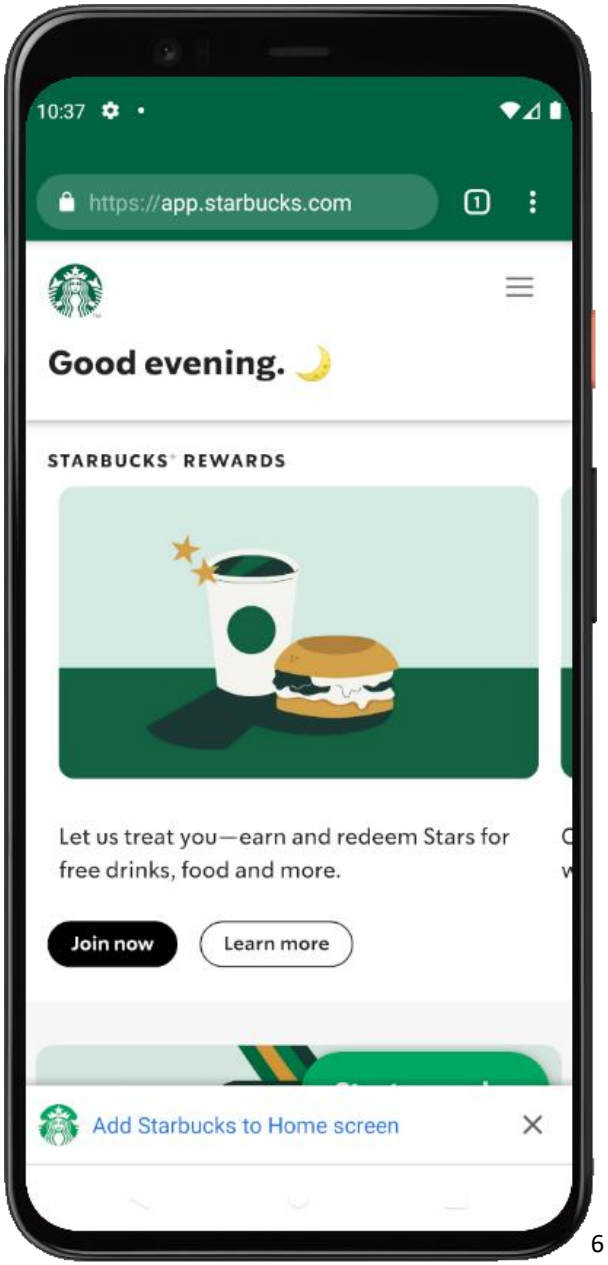
- Responsive Web app adapted to any mobile resolution
- Installable & can work on any platform
- Experience feels like a native app

✓ Can work offline, provide access to limited OS services such as GPS, push notifications

– Slower performance (Run inside a WebView)

– Least access to hardware, sensors, OS

– Can't Download on the app stores



Hybrid-Web App Development

- Hybrid-Web Apps: apps blend
  - mobile-optimized UI components (written using HTML, CSS, and JavaScript) with
  - native modules or **bridge plugins** for accessing Camera, Geolocation, Bluetooth and other services

✓ Lower development costs (Single codebase)

✓ Multiplatform - Write once, run anywhere

✓ Downloadable from app stores

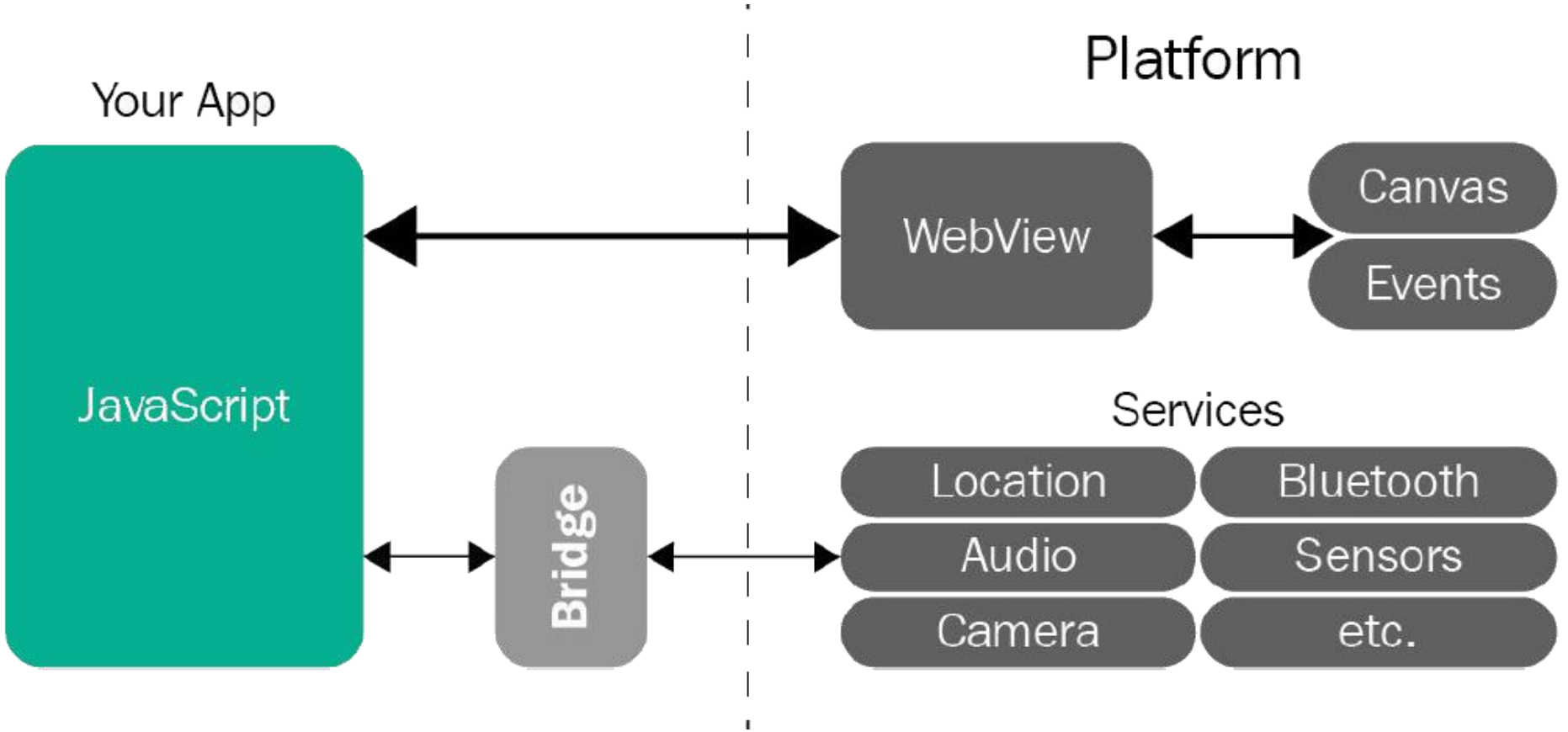
– Slower performance (not suitable for 3D games and other performance-intensive applications)

– Highly dependent on libraries and frameworks

APACHE CORDOVA™

Web / Hybrid-Web App Platforms

- App runs inside a **WebView** responsible for UI Rendering.
- App access the platform services via a **bridge**



https://wajahatkarim.com/2019/11/how-is-flutter-different-from-native-web-view-and-other-cross-platform-frameworks/

Hybrid Cross-platform Native App Development

- Hybrid Cross-platform Native Apps written using React Native (JavaScript) that generates **native UI** elements or Flutter (Dart) that uses a **native rendering engine**

✓ Lower development costs (shared codebase)

✓ Leverage existing skillset (JavaScript, React, Dart)

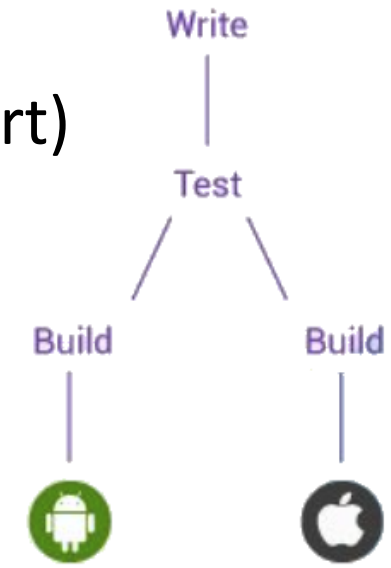
✓ Multiplatform utilizing a single codebase

✓ UI performance is almost as fast as native

✓ Downloadable from app stores

– Highly dependent on libraries and frameworks

– Delayed to update to latest native APIs

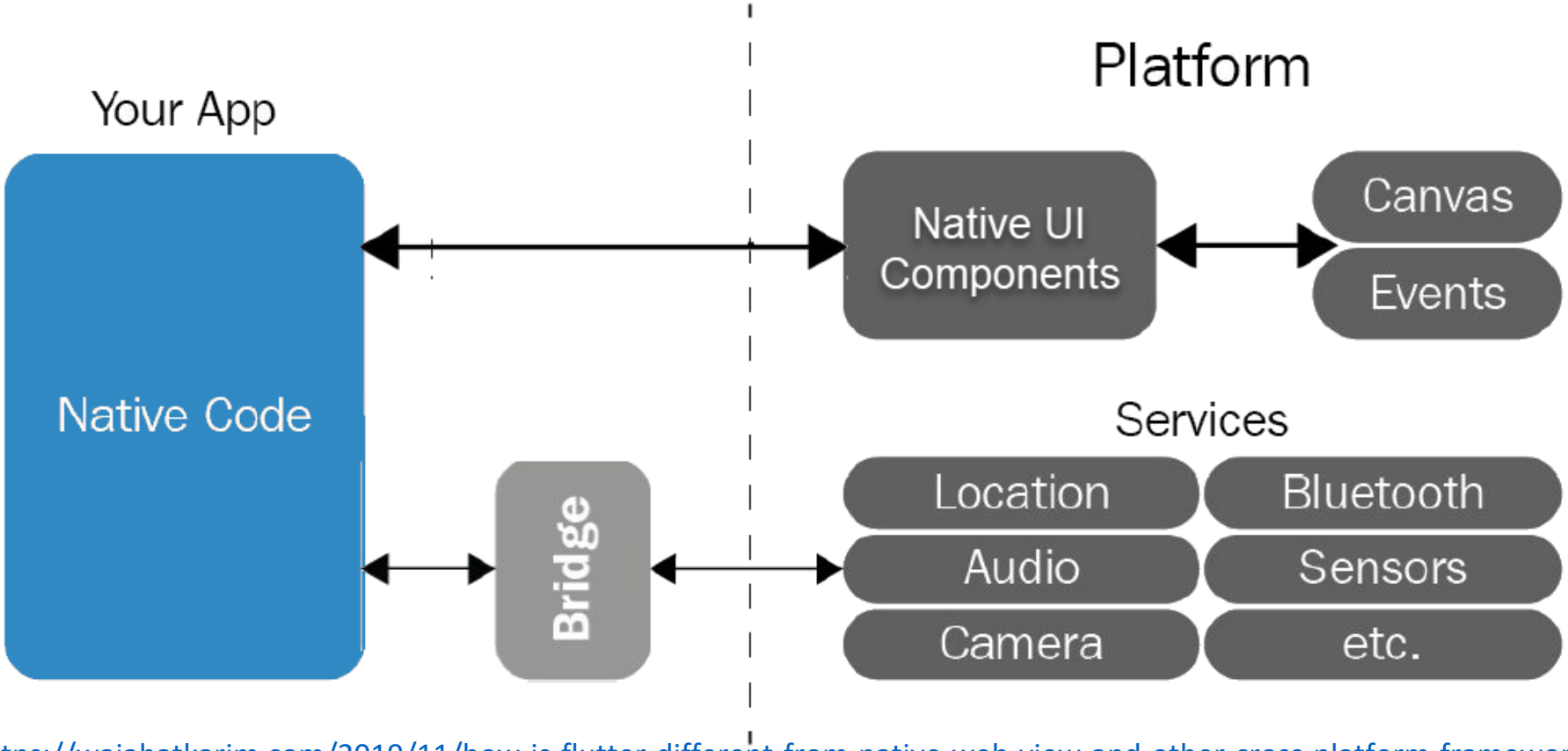


Flutter

React Native

React Native Hybrid Cross-platform Native App Platform

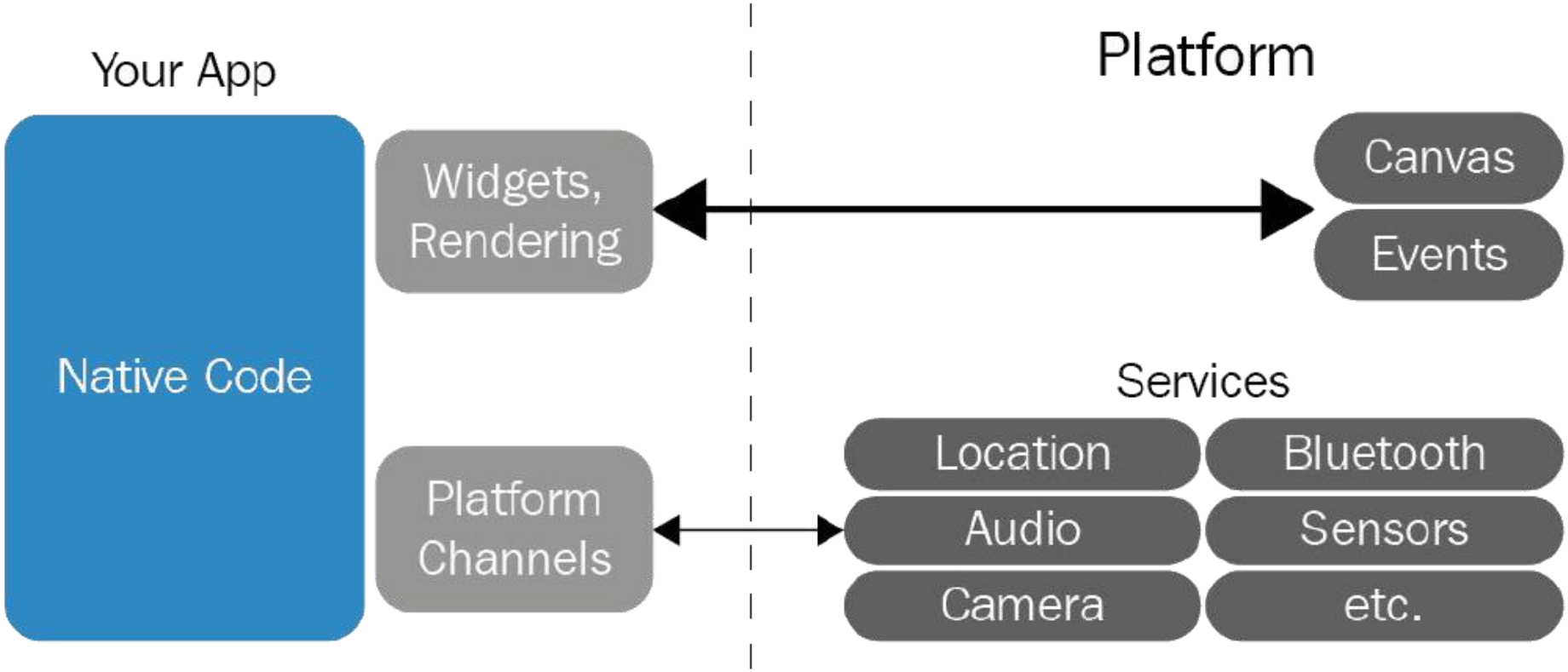
- React Native compiles JavaScript UI components into equivalent **native UI** elements (remaining code doesn't get compiled, instead runs in a separate JavaScript thread)
- App access the platform services via a **bridge**



https://wajahatkarim.com/2019/11/how-is-flutter-different-from-native-web-view-and-other-cross-platform-frameworks/

Flutter

- Flutter App is **compiled into native code**, UI uses Flutter own custom widgets rendered by the framework's **graphics engine** (https://skia.org/) to work across devices.
- App uses Platform Channels to access the platform services



https://wajahatkarim.com/2019/11/how-is-flutter-different-from-native-web-view-and-other-cross-platform-frameworks/

Native App Development

- Uses platform-specific (Android/iOS) UI components, languages and technologies

✓ Access to all native APIs, hardware, sensors, & OS

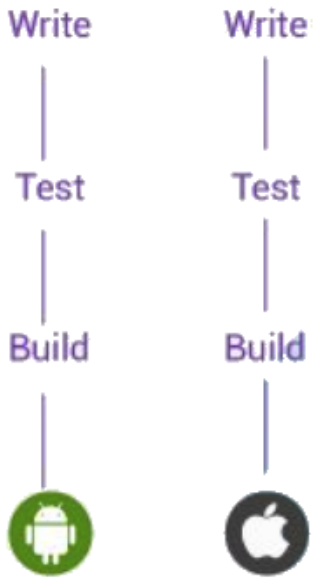
- No third-party dependencies

✓ Run directly on OS: Fast performance

✓ High-quality User Experience (UX)

– No codebase reuse

– High dev cost and longer time to market: requires **multiple code bases** and teams

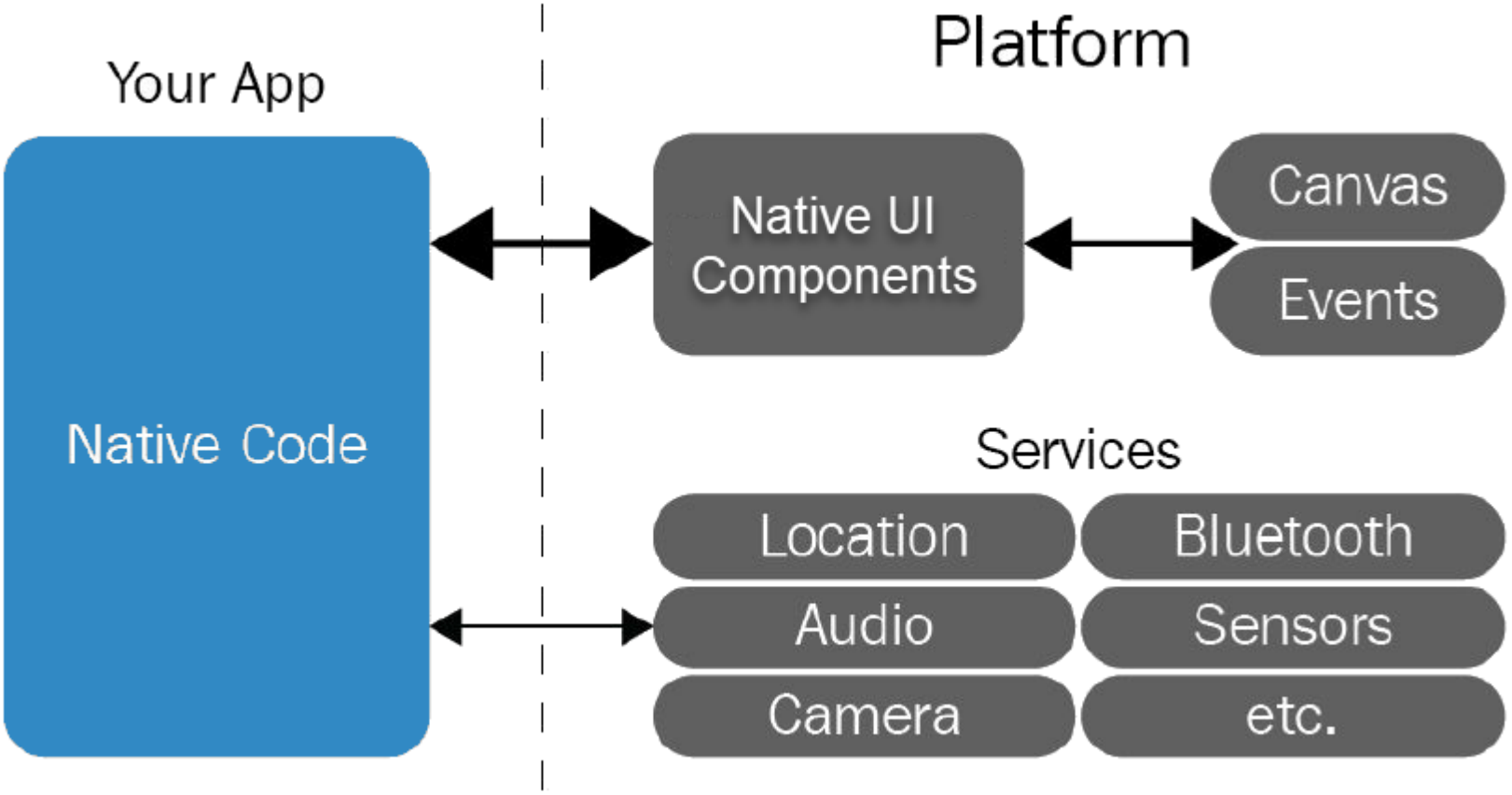


android

iOS

Native Android/iOS Platforms

The **app** has direct access to the platform services



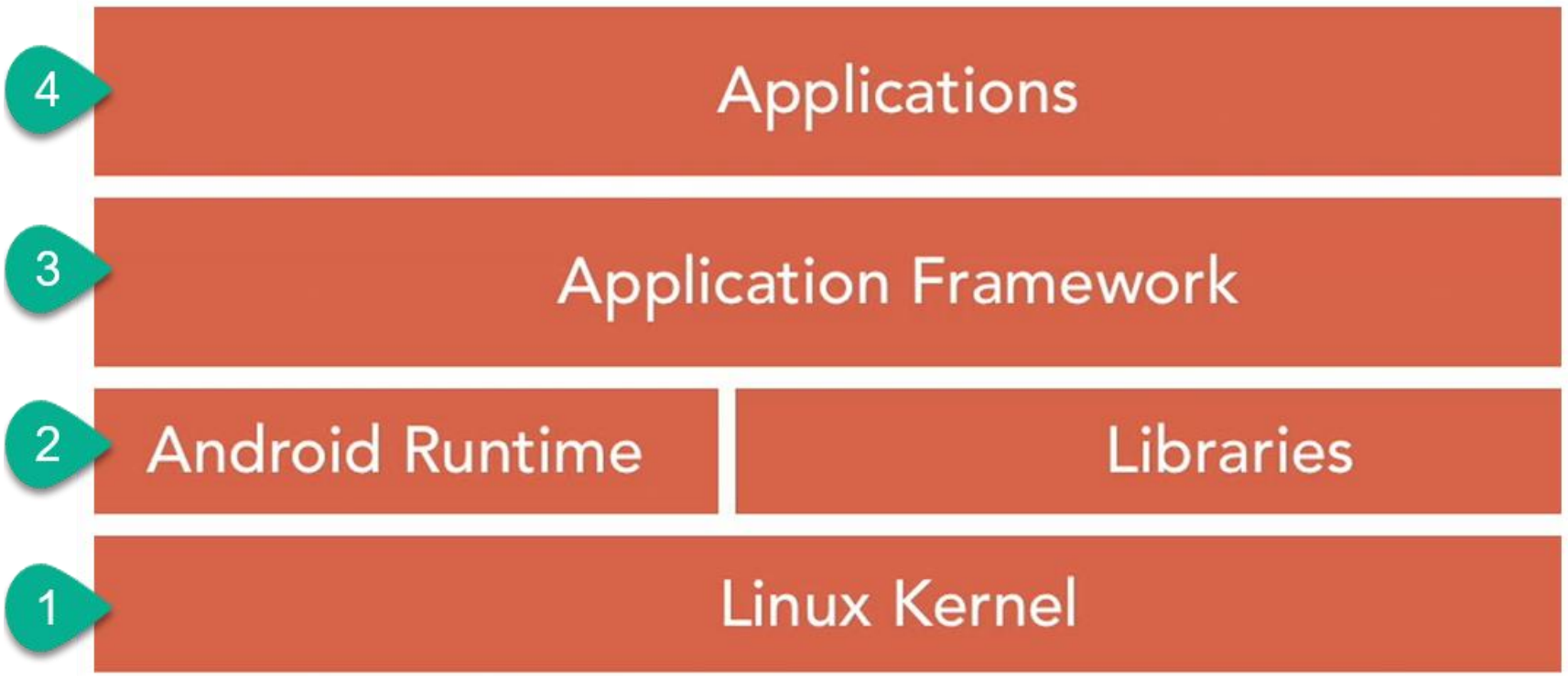
https://wajahatkarim.com/2019/11/how-is-flutter-different-from-native-web-view-and-other-cross-platform-frameworks/

Introduction to Android

What is Android?

- Open source mobile operating system (OS) based on Linux kernel for phones, tablets, wearable
  - originally purchased by Google from Android, Inc. in 2005
- The #1 OS worldwide
  - Used on over 80% of smartphones
  - As of 2019, over 2.5 billion Android devices worldwide
  - Over 2 Million Android apps in Google Play store
- Highly customizable for devices by vendors

Android Software Stack



1. Interacts and manages hardware
2. Expose native APIs & run apps
3. Java API exposing Android OS features
4. System and user apps (e.g., contacts, camera)

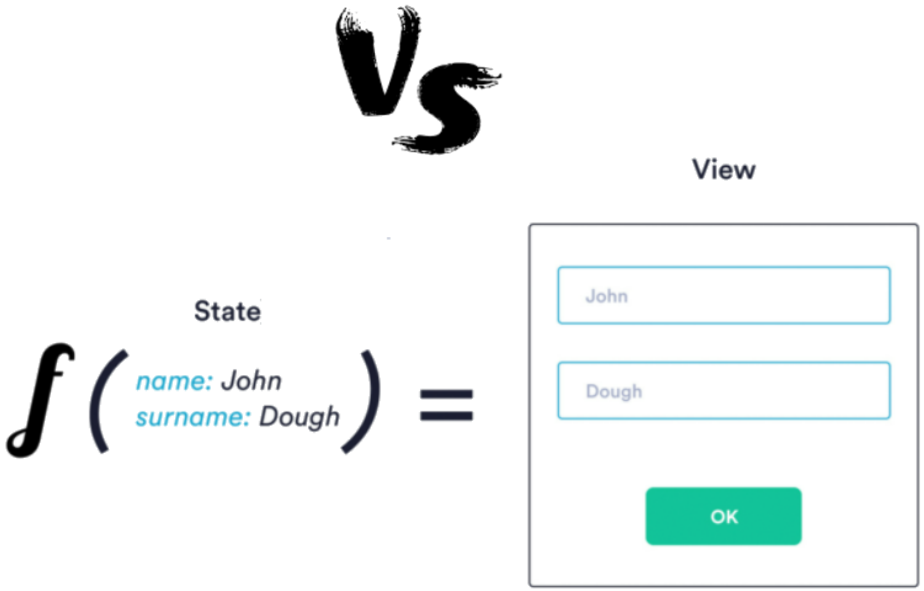
Android Software Stack

1. Optimized **Linux Kernel** for interacting with the device's processor, memory and hardware drivers (e.g., WiFi Driver)
   - Acts as an abstraction layer between the hardware and the rest of the software stack
2. **Android runtime (ART)** = Virtual Machine to run Apps
   - Each app runs in its own process and with its own instance of the Android Runtime that controls the app execution (e.g., permission checks) in isolation from other apps
   - Expose native APIs and OS Core Libraries including 2D/3D graphics, Audio Manager, SQLite database, encryption …
3. **Application Framework**: Java APIs (Application Programming Interfaces) make Android OS features available to Apps (e.g., Activity Manager that manages the lifecycle of apps)

https://developer.android.com/guide/platform

Imperative UI vs. Declarative UI

```
TextView greetings = (TextView) findViewById(R.id.tv_greeting)
greetings.text = “Hello world.”
```



Imperative UI vs. Declarative UI

👑 In **Imperative UI**, the steps to create the UI are **explicitly and fully** defined and then it is updated using methods / properties of the UI elements

- To change the view the developer, need to specify <u>when</u> to change and <u>how</u> to change the view

📣 In **Declarative UI**, Describe what the Ul should look like & the state data to feed to the UI

- The UI runtime has the responsibility to <u>observe</u> the state changes then <u>automatically update</u> the UI to reflect state changes

Imperative vs. Declarative UI

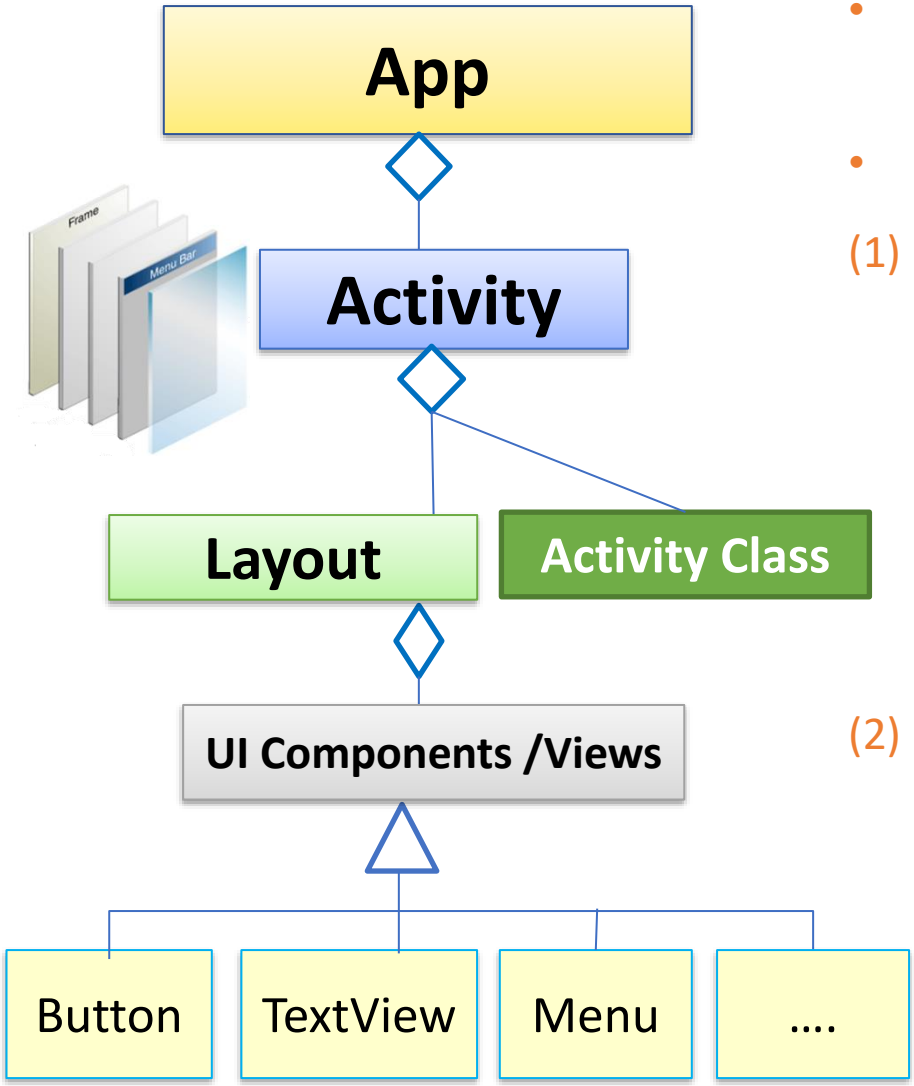
Imperative:

- Lots of boilerplate and boring code
- Errors and bugs prone
- Hard to maintain

Declarative: Describe WHAT to see NOT HOW

- ✓ Less code to write → Fewer bugs and more flexible
- ✓ State changes trigger automatic update of the UI to reflect state changes
- ✓ Improves UI reusability

Imperative UI - old Android Programming Model



- App is composed of one or more **screens** (called **Activity**)
- An **activity** has:

(1) a **Layout** that define its appearance (how it **looks like**)

- Layout acts as a **container** for UI Components (called **View**)
- It decides the size and position of views placed in it

(2) Activity Kotlin class that provides the data to the UI and handles events

- UI Components **raise Events** when the user interacts with them (such as a Clicked event is raised when a button is pressed).
- In the activity class we define **Event Handlers** to respond to the UI events

Imperative UI - Activity

- **Activity** provides the UI that the user interacts with
  - Allow the user to do something such as order groceries, send email
  - Has **layout** (.xml) file & **Activity class**
  - This allows a **good separation** between the UI and the app logic
- Connecting activity with the layout is done in the **onCreate** method

```
setContentView(R.layout.activity_main)
```

- Activity class defines listeners to handle events:
  - User interaction events such press a button or enters text in a text view
  - External events such as receiving a notification or screen rotation

Imperative UI - Example

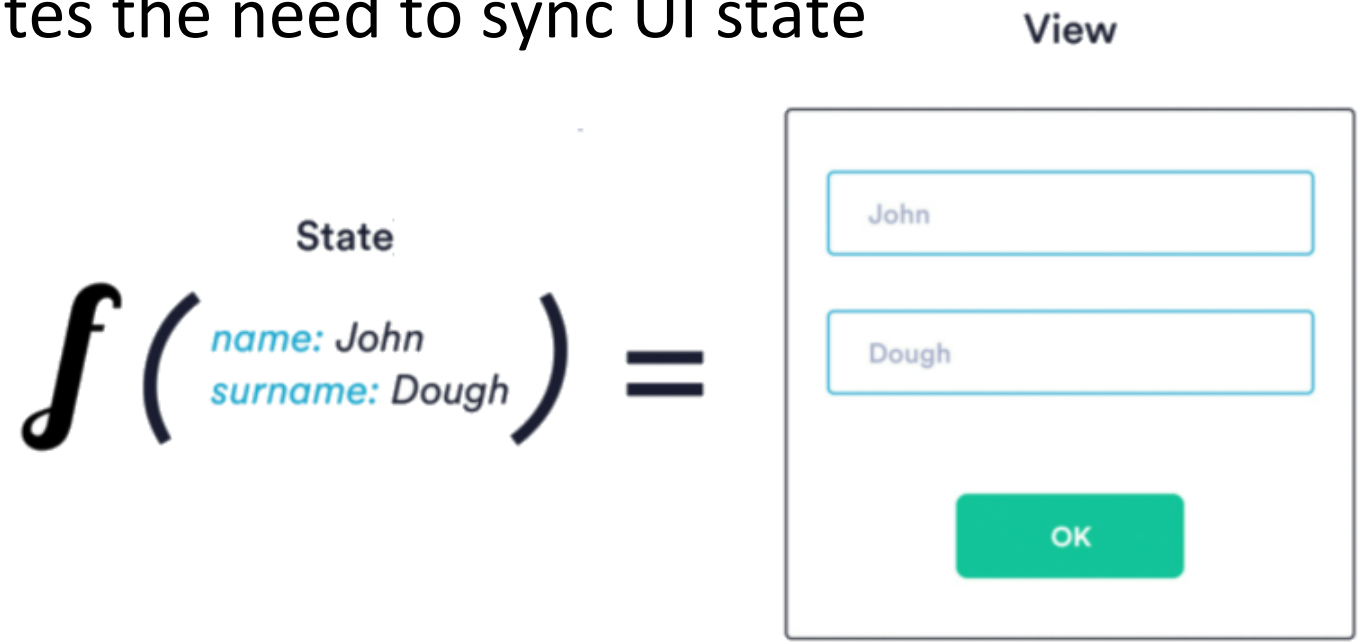
```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        changeColorBtn.setOnClickListener {
            greetingTv.setTextColor(getRandomColor())
        }
    }
```

Connects activity with layout

Declarative UI

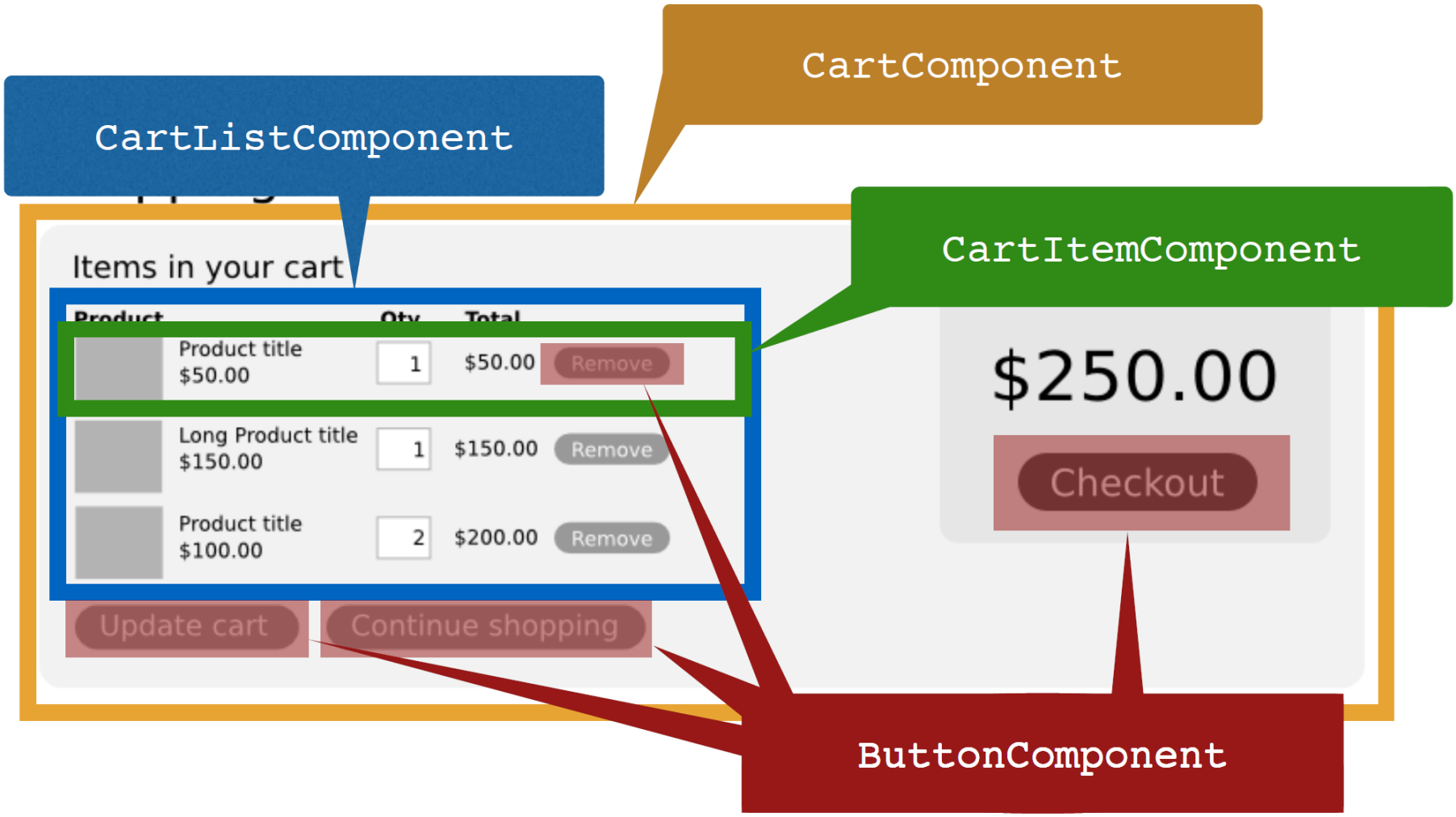
- Describe what elements you need in your UI and to a degree what they should look like
- **UI = f(state)** : UI is a visual representation of state
- State changes trigger automatic update of the UI
  - Eliminates the need to sync UI state



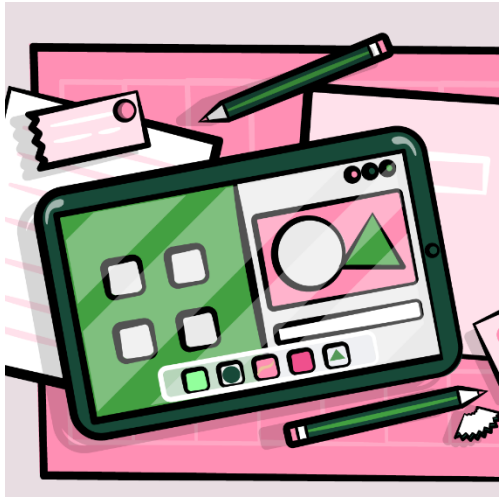
Mobile App UI Design Process

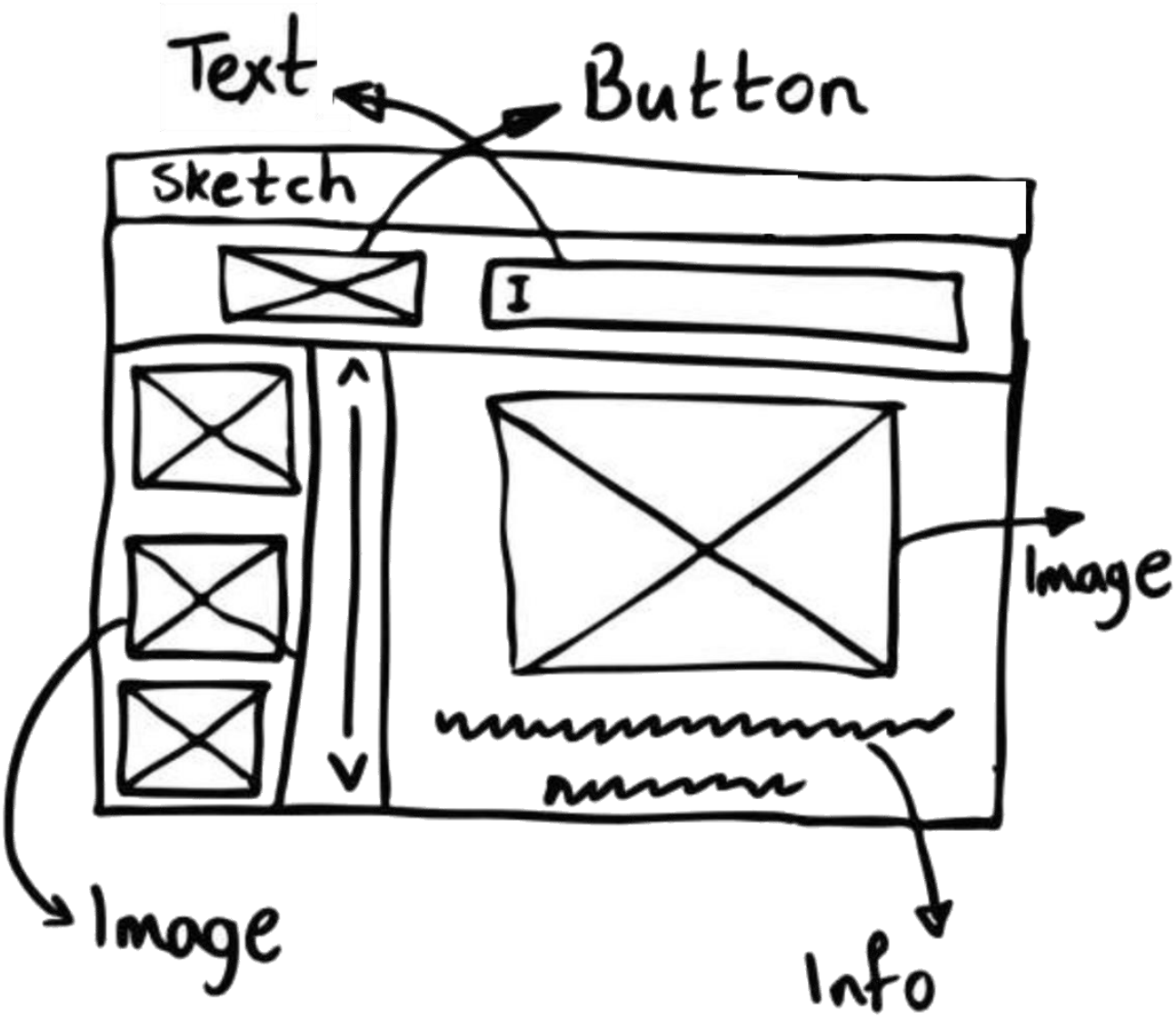
1. Design it on paper (sketch)
   - Decide what information to present to the user and what input they should supply
   - Decide the UI components and the layout on paper
2. Breakdown the UI into small reusable UI components (building blocks)
3. Identify the state and events per UI component
4. Compose the screen from building block components using appropriate layouts
5. Add event handlers to respond to the user actions
   - Do something when the user presses a button, selects an item from list, change text of input field, etc.

App UI = a tree of components



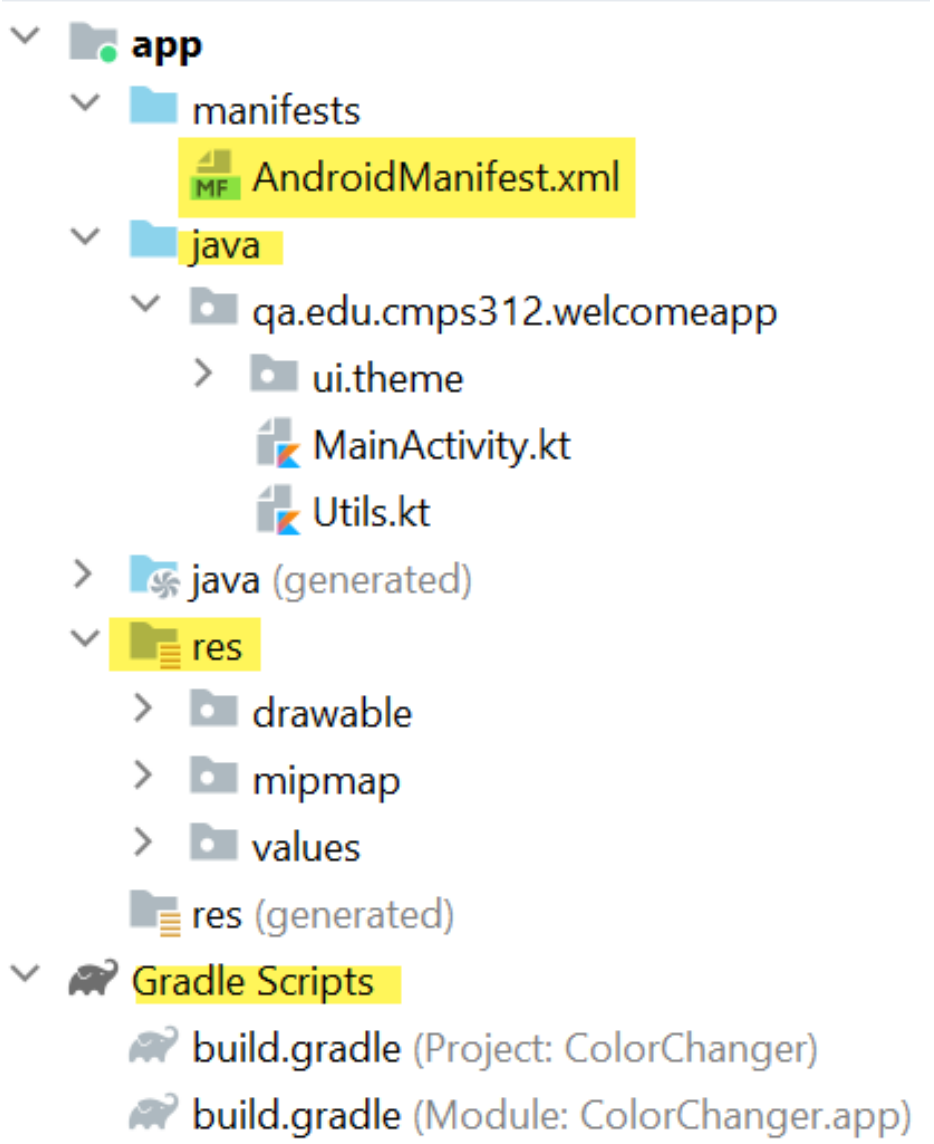
UI Sketch - Example





You may design different layouts per screen size

Android Project structure



- **AndroidManifest.xml**
  - app config and settings (e.g., list app activities and required permissions)
- **java**/...
  - Kotlin source code
- **res**/... = resource files *(many are **XML**)*
  - drawable/ = images
  - Mipmap = app/launcher icons
  - values/ = Externalize constant values
- **Gradle**
  - a build/compile management system
  - **build.gradle** = define config and dependencies (one for entire project & other for app module)

Resources

- Comparing Cross-Platform Frameworks
  - https://ionic.io/resources/articles/ionic-vs-react-native-a-comparison-guide
- Android Kotlin Fundamentals Course
  - https://codelabs.developers.google.com/android-kotlin-fundamentals/
  - https://developer.android.com/courses/android-basics-kotlin/course
- Android Dev Guide
  - https://developer.android.com/guide/